Chapter 7

Expressions and
Assignment Statements

Topics

- Introduction
- Arithmetic Expressions
- Infix, Prefix and Postfix
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

- Expressions are the fundamental means of specifying computations in a programming language
  - In imperative languages expressions are the right hand side of assignment statements
  - In functional languages computation is simply expression evaluation
- To understand expression evaluation, we need to be familiar with the orders of operator and operand evaluation
  - May be only partially specified by associativity and precedence rules
  - If not completely specified we might get different results in different implementations

Introduction

- Other issues are type mismatches, coercions and short-circuit evaluation
- The essence of imperative languages is the dominant role of assignment statements that change the values of memory cells

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls

Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

Operator Arity

- A unary operator has one operand
- A binary operator has two operands
- A ternary operator has three operands

Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which adjacent operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -
  - Relational Operators

Comparison of Precedence

Operator	C-like	Ada	Fortran
Unary -	7	3	3
**	n/a	5	5
* /	6	4	4
+ -	5	3	3
== !=	4	2	2
< <= ...	3	2	2
not	7	2	2

Associativity

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different; all operators have equal precedence and all operators associate right to left
- Smalltalk: binary methods that we see as operators have equal precedence and left associativity
- Precedence and associativity rules can be overriden with parentheses

Associativity of Operators

Lang	+ - * /	Unary -	**	== != < ...
C-like	L	R	n/a	L
Ada	L	non	non	non
Fortran	L	R	R	L
VB	L	R	L	non

- Note that left associative relational operators allow expressions such as a < b < c
  - But in C this means
    - `if (a < b) then (1 < c) else (0 < c)`
  - Not
    - `(a < b) && (b < c)`
- With non-associative relational operators expression such as a < b < c are not legal

Associativity

- For +,* operators that have the associative property optimizing compilers may reorder expression evaluation
  - In theory x * y * z * w can be evaluated in any order
  - But if x and z are very large and y and w are very small we can get a different results from
    ((x * y) * z) * w ((x * z) * y) * w
  - In floating point arithmetic we can lose precision or even produce infinities
  - In integer arithmetic we have overflow or wraparound
  - We could specify order of evaluation with parentheses
    (x * y) * (z * w)

Ruby and Smalltalk Expressions

- All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods
  - One result of this is that these operators can all be overridden by application programs

Ternary Conditional Operator

- Conditional Expressions
  - In most C-like languages (C, C++, Java, PHP, Javascript, …)

```
average = (count == 0) ? 0 : sum / count
```

Same as this code:

```
if (count == 0)
   average = 0
else
   average = sum /count
```

Some languages do not require parentheses:

```
average = count == 0 ? 0 : sum / count
```

Operand Evaluation Order

- Operand evaluation order
  1. Variables: fetch the value from memory
  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
  3. Parenthesized expressions: evaluate all operands and operators first
- Evaluation order is generally irrelevant except when an operand is a function call that has side effects

Side Effects

- Side effects occur when:
  - A function changes one of its parameters
  - A function changes a non-local variable
  - A function performs input or output
- Example

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

Side Effects

- Changing a non-local variable

```
int a = 5;
int func(x){
      a = 42;
      return x % a;
}
void main(){
      a = a + func(84);
}
```

- Is the value of a 7 or 44?

Functional Side Effects

- Two possible solutions to the problem
  1. Write the language definition to disallow functional side effects
     - No two-way parameters in functions
     - No non-local references in functions
     - **Advantage:** it works!
     - **Disadvantage:** inflexibility of one-way parameters and lack of non-local references
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations
     - Java requires that operands appear to be evaluated in left-to-right order
     - C and C++ do not require any fixed order

Side Effects

- The generally accepted rule for programming is that value returning functions should not have side-effects
- Less generally accepted is the notion that procedures should not have side effects except by modifying one or more arguments
- But most imperative and OO languages have no mechanisms to enforce side-effect rules

Referential Transparency

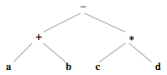
- An expression has referential transparency if it can be replaced with its value without changing the action of the program

```
ans1 = (fun(a)+b) / (fun(a)+c);
Temp = fun(a)
ans2 = (temp+b) / (temp+c);
```

- Absence of functional side effects is neccesary (but not sufficient) for referential transparency
- We will discuss further with functional languages

Infix Expression Semantics

- Most programming languages use infix notation
- Infix is inherently ambiguous unless associativity and precedence are defined
- Ex a + b – c * d usually means (a + b) – (c * d)
- In Smalltalk it means ((a + b) – c) * d
- In APL it means a + (b – (c * d))

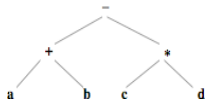


Prefix and Postfix notations

- Two different ways to represent expressions; both are unambiguous
  - Infix (a + b) - (c * d)
  - Polish Prefix: - + a b * c d
  - Polish Postfix: a b + c d * -
    - Also know as Reverse Polish Notation or RPN
    - Introduced early 20th century by Polish mathematician Jan Lukasiewicz
  - Cambridge Polish: (- (+ a b) (* c d))
- Infix uses associativity and precedence to disambiguate.

Obtaining Prefix and Postfix Forms

- Both forms can be obtained by traversing expression trees
  - Prefix walk or preorder traversal
    1. Generate the value of the node
    2. Visit the left subtree
    3. Visit the right subtree
  - Postfix walk or postorder traversal
    1. Visit the left subtree
    2. Visit the right subtree
    3. Generate the value of the node



Evaluation of RPN Expressions

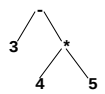
- Uses a stack:

1. Get next token (operator or operand) from input stream
2. If the token is an operand
   push it on the stack
   else // an n-ary operator
   pop the top n operands from the stack (R to L)
   perform the operation
   push result on top of the stack
3. Repeat 1-2 until EOF
4. Pop final result off the stack

RPN Example

```
Input: 3 4 5 * -  (Infix 3 - 4 * 5)

Token Action          Stack
3   Push              (3)
4   Push              (3 4)
5   Push              (3 4 5)
*   Pop 5, Pop 4;
    Push 4*5 = 20     (3 20)
-   Pop 20, Pop 3
    Push 3-20 = -17   (-17)
EOF Pop and return -17
```



Example 2 `Input: 2 3 * 12 3 / + 5 3 * 6 - +`

```
TokenAction          Stack
2    Push                (2)
3    Push                (2 3)
*    Pop 3, Pop 2;
     Push 2 * 3 = 6      (6)
12   Push                (6 12)
3    Push                (6 12 3)
/    Pop 3, Pop 12;
     Push 12/3 = 4       (6 4)
+    Pop 4, Pop 6
     Push 6+4 = 10       (10)
5    Push                (10 5)
3    Push                (10 5 3)
*    Pop 3, Pop 5;
     Push 5*3 = 15       (10 15)
6    Push                (10 15 6)
-    Pop 6, Pop 15
     Push 15-6 = 9       (10 9)
+    Pop 9, Pop 10
     Push 10+9 = 19      (19)
EOF  Pop and return 19
```

Unary Operators

- Using Polish notation it is not possible to have the same operator for both unary and binary operations – e.g. the binary and unary minus
- Two solutions:
  - Use Cambridge Polish (parenthesized)
  - Use a different symbol (e.g, ~)
- With Cambridge Polish notation, operators such as + and – can be used as n-ary operators

  (+ a b c d) is a + b + c + d

Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*
- Some are common (e.g., + for `int` and `float`)
- Some are potential trouble (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability
- C++, C#, Ada allow user-defined overloaded operators

Overloaded Operators

- A design mistake in Javascript: using + for addition as well as string concatenation

```
var x = "10"
var y = x + 5;     // y is 105
Var z = x – 3;     // z is 7
```

Type Conversions

- A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., `float` to `int`
- A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`
- Note that widening conversion can lose precision, but the magnitude is retained

Type Conversions: Mixed Mode

- A *mixed-mode expression* is one that has operands of different types
- A *coercion* is an implicit type conversion
- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In Ada, there are virtually no coercions in expressions

Explicit Type Conversions

- Also known as casts or type casts
- Examples

```
C: (int)angle
Ada: Float (Sum)
```

- Ada syntax (and that of many other languages) looks like a function call

Type Conversions: Errors in Expressions

- Causes
  - Inherent limitations of arithmetic
    e.g., division by zero
  - Limitations of computer arithmetic
    e.g. overflow
- Often ignored by the run-time system

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages (`!=`, `/=`, `~=`, `.NE.`, `<>`, `#`)
- JavaScript and PHP have two additional relational operator, `===` and `!==`
  - Similar to their cousins, `==` and `!=`, except that they do not coerce their operands

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators

FORTRAN 77	FORTRAN 90	C	Ada
`.AND.`	`and`	`&&`	`and`
`.OR.`	`or`	`||`	`or`
`.NOT.`	`not`	`!`	`not`
			`xor`

No Boolean Type in C

- C89 has no Boolean type--it uses `int` type with 0 for false and nonzero for true
- One odd characteristic of C's expressions: **`a < b < c`** is a legal expression, but the result is not what you might expect:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., **`c`**)

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators
- Example: `if (x > y && y > z)`
  If x <= y, there is no need to evaluate `y > z`

Short Circuit Evaluation

- Without short-circuit evaluation some code can be problematic

```
Node p = head;
while (p != null && p.info != key)
        p = p.next;
if (p == null) // not in list
        ...
else // found it
        ...
```

  - If p is null then p.info will raise an exception

Short Circuit Evaluation

- We need more complex code

```
boolean found = false;
while (p != null && ! found) {
        if (p.info == key)
                found = true;
        else
                p = p.next;
}
```

Short Circuit Evaluation

- But with short circuit evaluation we have a potential problem with functions that have side effects

```
if f(a, b) && g(y) {
   /* do something */
}
```

- if f(a, b) returns false g never is called
- We also have a problem with side effects in expressions

```
if ((a > b) || (b++ / 3)){ }
```

Short Circuit Evaluation (continued)

- It is important to know if your language supports it
- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**&&** and **||**)
- Ada, VB.NET: programmer can specify either (short-circuit is specified with **and then** and **or else**)
- If your language supports it you can tests to guard against problems

```
if (count != 0  && total / count > 10)

if (isset($_GET['id'])&& len($_GET['id'])> 10)
```

Assignment Statements

- The general syntax
  `<target_var> <assign_operator> <expression>`
- The assignment operator
  - = FORTRAN, BASIC, the C-based languages
  - := ALGOLs, Pascal, Ada
- = can be hard to read when it is overloaded for the relational operator for equality
- Basic, Pascal
  `x = total = 0`
- The total=0 expression produces a Boolean that is assigned to x

Conditional Assignment Targets

- Conditional targets (Perl)
  `($flag ? $total : $subtotal) = 0`
- equivalent to

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```

Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C and all later curly brace languages, Visual Basic
  `a = a + b;` is written as `a += b`
- Compound assignment can be used with almost any binary operator

```
a += b;  a /= b;  a &&= b;  a ||= b;
n *= -1;  yy <<= 1;

$s .= "string concatenation in PHP" ;
```

Unary Assignment Operators

- Most curly brace languages have unary pre- and post- operators ++ and –-
  - From a high level perspective these are assignment operators but they the machine level INC and DEC
  - Originally designed in C
- Examples

```
sum = ++count;   //inc count then add to sum
sum = count++;   //add to sum then inc count
count--;         //dec count, same as –-count;
n = -count++;    // same as – (count++)
x = *p++;        // inc pointer p after dereference
x = (*p)++;      // dereference then inc value
```

Assignment Expressions

- In C, C++, and Java, the assignment statement produces a result and can be used as operands
- Example

```
while ((ch = getchar())!= EOF){…}
```

- And

```
void strcpy (char *s, char *t){
  while (*s++ = *t++)
  ;
}
```

Assignment Expressions

- Note that precedence of the assignment operator = is very low so assignment expressions usually have to be parenthesized
- Assignment expressions are a form of side-effect.
- Can lead to difficult to understand code
  `a = b + (c = d / b) -1;`
- But it's also useful for multiple assignment:
  `total = subtotal = count = subcount = 0;`
  - Note that = has to be right-associative for this to work

List Assignments

- Perl and Ruby support list assignments Ex:

```
($first, $second, $third) = (20, 30, 40);
```

- Note that this will "swap" variables

```
($second, $third) = ($third, $second);
```

- List assignment is a handy shortcut but can have some pitfalls. From the PHP manual:

  List() assigns the values starting with the right-most parameter. If you are using plain variables, you don't have to worry about this. But if you are using arrays with indices you usually expect the order of the indices in the array the same you wrote in the list() from left to right; which it isn't. It's assigned in the reverse order.